Exploration of the scalability of LocFaults approach for error localization with While-loops programs

Mohammed Bekkouche

Exploration of the scalability of LocFaults approach for error localization with While-loops programs

Mohammed Bekkouche
University of Nice-Sophia
Antipolis, I3S/CNRS
BP 121, 06903 Sophia
Antipolis Cedex, France
bekkouch@i3s.unice.fr

ABSTRACT

A model checker can produce a trace of counterexample, for an erroneous program, which is often long and difficult to understand. In general, the part about the loops is the largest among the instructions in this trace. This makes the location of errors in loops critical, to analyze errors in the overall program. In this paper, we explore the scalability capabilities of `LocFaults`, our error localization approach exploiting paths of CFG(Control Flow Graph) from a counterexample to calculate the MCDs (Minimal Correction Deviations), and MCSs (Minimal Correction Subsets) from each found MCD. We present the times of our approach on programs with *While*-loops unfolded b times, and a number of deviated conditions ranging from 0 to n. Our preliminary results show that the times of our approach, constraint-based and flow-driven, are better compared to `BugAssist` which is based on SAT and transforms the entire program to a Boolean formula, and further the information provided by `LocFaults` is more expressive for the user.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and features**]: Constraints; D.2.5 [**Testing and Debugging**]: Debugging aids, Diagnostics, Error handling and recovery

General Terms

Verification, Algorithms, Experimentation

Keywords

Error localization, LocFaults, BugAssist, Off-by-one bug, Minimal Correction Deviations, Minimal Correction Subsets

1. INTRODUCTION

Errors are inevitable in a program, they can harm proper operation and have extremely serious financial consequences. Thus it poses a threat to human well-being [17]. This link [3] cites recent stories of software bugs. Consequently, the debugging process (detection, localization and correction of errors) is essential. The location of errors is the step that costs the most. It consists of identifying the exact locations of suspicious instructions [18] to help the user to understand why the program failed, which facilitates him in the task of error correction. Indeed, when a program P is not conformed with its specification (P contains errors), a model checker can produce a trace of a counterexample, which is often long and difficult to understand even for experienced programmers. To solve this problem, we have proposed an approach [5] (named `LocFaults`) based on constraints that explores the paths of CFG (Control Flow Graph) of the program from the counterexample, to calculate the minimal subsets to restore the program's compliance with its postcondition. Ensuring that our method is highly scalable to meet the enormous complexity of software systems is an important criterion for its quality [9].

Different statistical approaches for error localization have been proposed; e.g.: `Tarantula` [11] [10], `Ochiai` [1], `AMPLE` [1], `Pinpoint` [6]. The most famous is `Tarantula`, which uses different metrics to calculate the degree of suspicion of each instruction in the program while running a battery of tests. The weakness of these approaches is that they require a lot of test cases, while our approach uses one counterexample. Another critical point in statistical approaches is that they require an oracle to decide if the result of a test case is correct or not. To overcome this problem, we consider the framework of Bounded Model Checking (BMC) which only requires a postcondition or assertion to check.

The idea of our approach is to reduce the problem of error localization to the one which is to compute a minimal set which explains why a CSP (Constraint Satisfaction Problem) is infeasible. The CSP represents the union of constraints of the counterexample, the program, and the assertion or the postcondition violated. The calculated set can be a MCS (Minimal Correction Subset) or a MUS (Minimal Unsatisfiable Subset). In general, test the feasibility of a CSP over a finite domaine is a NP-complete problem (intractable)[1], one of the most difficult NP problems. This means, explaining the infeasibility in a CSP is as hard or more (it can be classified as NP-hard problem). `BugAssist` [13] [12] is a BMC method of error localization using a Max-SAT solver to calculate the merger of MCSs of the

[1]If this problem could be solved in polynomial time, then all NP-complete problems would be too.

ESEC/FSE'15 August 31-September 4, 2015, Bergamo, Italy.

Boolean formula of the entire program with the counterexample. It becomes inefficient for large programs. `LocFaults` also works from a counterexample to calculate MCSs.

In this paper, we explore the scalability of `LocFaults` on programs with *While*-loops unfolded b times, and a number of deviated conditions ranging from 0 to 3.

The contribution of our approach against BugAssist can be summarized in the following points:

* We do not transform the entire program in a system of constraints, but we use the CFG of the program to collect the constraints of the path of counterexample and paths derivatives thereof, assuming that at most k conditionals may contain errors. We calculate MCSs only on the path of counterexample and paths that correct the program;
* We do not translate the program instructions into a SAT formula, instead numerical constraints that will be handled by constraint solvers;
* We do not use MaxSAT solvers as black boxes, instead a generic algorithm to calculate MCSs by the use of a constraint solver;
* We limit the size of the generated MCSs and the number of deviated conditions;
* We can work together more solvers during the localization process and take the most efficient according to the category of CSP constructed. For example, if the CSP of the path detected is of type linear over integers, we use a MIP (Mixed Integer Programming) solver; if it is nonlinear, we use a CP (Constraint Programming) solver and/or as well as MINLP (Mixed Integer Nonlinear Programming).

Our practical experience has shown that all these restrictions and distinctions enable `LocFaults` to be faster and more expressive.

The paper is organized as follows. Section 2 introduces the definition of MUS and MCS. In Section 3, we define the problem $\leq k$-MCD. We explain a paper contribution for the treatment of erroneous loops, including the *Off-by-one* bug, in Section 4. A brief description of our `LocFaults` algorithm is provided in Section 5. The experimental evaluation is presented in Section 6. Section 7 talks about the conclusion and future work.

2. DEFINITIONS

In this section, we introduce the definition of an IIS/MUS and MCS.

CSP.

A CSP (Constraint Satisfaction Problem) P is defined as a triple $< X, D, C >$, where:

* X a set of n variables $x_1, x_2, ..., x_n$.
* D the tuple $< D_{x_1}, D_{x_2}, ..., D_{x_n} >$. The set D_{x_i} contains the values of the variable x_i.
* C=$\{c_1, c_2, ..., c_n\}$ is the set of constraints.

A *solution* for P is an instantiation of the variables $\mathcal{I} \in D$ that satisfies all the constraints in C. P is infeasible if it has no solutions. A sub-set of constraints C' in C is also said infeasible for the same reason except that it is limited to the constraints in C'.
We denote as:

- $Sol(< X, C', D >) = \emptyset$, to specify that C' has no solutions, so it is unfeasible.
- $Sol(< X, C', D >) \neq \emptyset$, to specify that C' has at least one solution, so it is feasible.

We say that P is *linear* and denote LP (Linear Program) iff all constraints in C are linear equations/inequalities, it is *continuous* if the domain all variables is real. If at least one of the variables in X is integer or binary (Special cases of an integer), and the constraints are linear, P is called a program *linear mixed* MIP (Mixed-integer linear program). If the constraints are nonlinear, we say that P is a program *nonlinear* NLP (NonLinear Program).

Let $P =< X, D, C >$ an infeasible CSP, we define for P:

IS.

An IS (Inconsistent Set) is an infeasible subset of constraints in the constraint set infeasible C. C' is an IS iff:

* $C' \subseteq C$.
* $Sol(< X, C', D >) = \emptyset$.

IIS or MUS.

An IIS (Irreducible Inconsistent Set) or MUS (Minimal Unsatisfiable Subset) is an infeasible subset of constraints of C, and all its strict subsets are feasible. C' is an IIS iff :

* C' is an IS.
* $\forall\ C'' \subset C'.Sol(< X, C'', D >) \neq \emptyset$, (each of its parts contributes to the infeasibility), C' is called irreducible.

MCS.

C' is a MCS(Minimal Correction Set) iff :

* $C' \subseteq C$.
* $Sol(< X, C \backslash C', D >) \neq \emptyset$.
* $\nexists\ C'' \subset C'$ such as $Sol(< X, C \backslash C'', D >) \neq \emptyset$.

3. THE PROBLEM $\leq K$-MCD

Given an erroneous program modeled in CFG[2] $G = (C, A, E)$: C is the set of conditional nodes; A is the set of assignment blocks; E is the set of arcs, and a counterexample. A MCD (*Minimal Correction Deviation*) is a set $D \subseteq C$ such as the propagation of the counterexample on all the instructions of G from the root, while having denied each condition[3] in D, allows the output to satisfy the postcondition. It is called minimal (or irreducible) in the sense that no element can be removed from D without losing this property. In other

[2]We use Dynamic Single Assignment (DSA) form [2] transformation that ensures that each variable is assigned only once on each path of the CFG.
[3]The condition is denied to take the branch opposite to that where we had to go.

words, D is a minimal program correctness in the set of conditions. The size of minimal deviation is its cardinal. The problem $\leq k$-*MCD* is to find all MCDs of size smaller or equal to k.

For example, the CFG of the program AbsMinus (see fig. 2) has one minimal size deviation 1 for the counterexample $\{i = 0, j = 1\}$. Certainly, the deviation $\{i_0 \leq j_0, k_1 = 1 \wedge i_0 \neq j_0\}$ corrects the program, but it is not minimal; only one minimal correction deviation for this program is $\{k_1 = 1 \wedge i_0 \neq j_0\}$.

```
1  class AbsMinus {
2  /*@ ensures
3    @ ((i<j)==>(\result==j-i
       ))&&
4    @ ((i>=j)==>(\result==i-
       j));*/
5    int AbsMinus (int i, int
       j){
6     int result;
7     int k = 0;
8     if (i <= j) {
9      k = k+2;//error:
         should be k=k+1
10    }
11    if (k == 1 && i != j) {
12     result = j-i;
13    }
14    else {
15     result = i-j;
16    }
17   }
18 }
```

Figure 1: The program AbsMinus

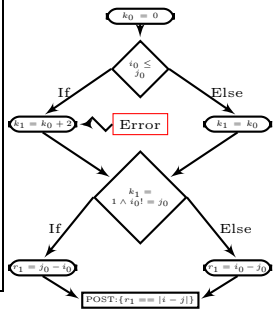


Figure 2: The CFG in DSA of AbsMinus

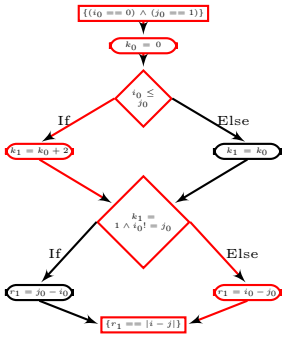


Figure 3: The path of the counterexample

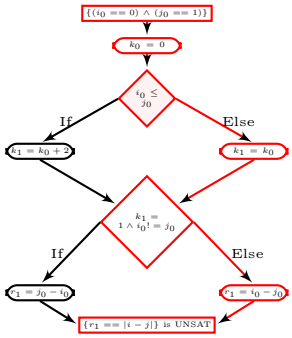


Figure 4: The path obtained by deviating the condition $i_0 \leq j_0$

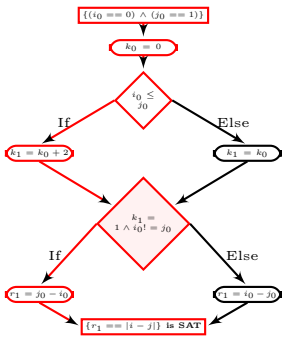


Figure 5: The path by deviating the condition $k_1 = 1 \wedge i_0! = j_0$

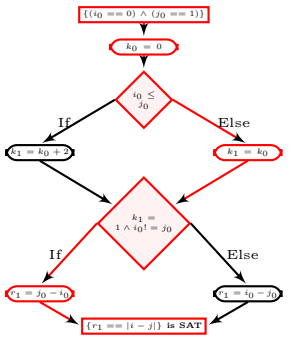


Figure 6: The path of a non-minimal deviation:$\{i_0 \leq j_0, k_1 = 1 \wedge i_0! = j_0\}$

Conditions deviated	*MCD*	*MCS*	Figure
$\emptyset$	/	$\{r_1 = i_0 - j_0 : 15\}$	fig. 3
$\{i_0 \leq j_0 : 8\}$	Non	/	fig. 4
$\{k_1 = 1 \wedge i_0! = j_0 : 11\}$	Oui	$\{k_0 = 0 : 7\}$, $\{k_1 = k_0 + 2 : 9\}$	fig. 5
$\{i_0 \leq j_0 : 8$, $k_1 = 1 \wedge i_0! = j_0 : 11\}$	Non	/	fig. 6

Table 1: The progress of `LocFaults` for the program AbsMinus.

The table 1 summarizes the progress of `LocFaults` for the program AbsMinus, with at most 2 conditions deviated from the following counterexample $\{i = 0, j = 1\}$.

We display the conditions deviated, if they are minimal deviation or non minimal, and the calculated MCSs from the constructed constraint system : see respectively the columns 1, 2 and 3. Column 4 shows the figure illustrating the path explored for each deviation. In the first and the third column we show in addition of the instruction, its line in the program. For example, the first line in the table shows that there is a single MCS found ($\{r_1 = i_0 - j_0 : 15\}$) on the path of the counterexample.

4. ERROR LOCALIZATION IN LOOPS

As part of Bounded Model Checking (BMC) for programs, unfolding can be applied to the entire program or it can be applied to loops separately [9]. Our algorithm LocFaults [4] [5] for error localization is placed in the second approach; that is to say, we use a bound b to unfold loops by replacing them with conditional statements nested of depth b. Consider for instance the program Minimum (see fig. 7), containing a single loop, that calculates the minimum in an array of integers. The effect on control flow graph of the program Minimum before and after unfolding is illustrated in Figures 7 and 8 respectively. The *While*-loop is unfolded 3 times, as 3 is the number of iterations needed for the loop to calculate the minimum value in an array of size 4 in the worst case.

`LocFaults` takes as input the CFG of the erroneous program, CE a counterexample, b_{mcd}: a bound on the number of deviated conditions, b_{mcs}: a bound on the size of MCSs calculated. It allows to explore the CFG in depth by diverting at most b_{mcd} conditions from the path of the counterexample:

* It propagates CE on the CFG until the postcondition. Then it calculates the MCSs on the CSP of the path generated to locate errors on the path of counterexample.

* It seeks to enumerate the sets $\leq b_{mcd}$-MCD. For each found MCD, it calculates the MCSs on the path that arrives at the last deviated condition and allows to take the path of the deviation.

Among the most common errors associated with loops according to [14], the *Off-by-one* bug, i.e. loops that iterate one too many or one too few times. This may be due to improper initialization of the loop control variables, or an erroneous condition of the loop. The program *Minimum* presents a case of this type of error. It is erroneous because of its loop *While*, the falsified instruction is on the condition of the loop (line 9): the correct condition should be $(i < tab.length)$ ($tab.length$ is the number of elements of the table tab). From the following counterexample

$PATH$	MCSs
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0], min_0 = tab_0[0], i_0 = 1, min_1 = tab_0[i_0], i_1 = i_0 + 1, min_2 = tab_0[i_1], i_2 = i_1 + 1, min_3 = min_2, i_3 = i_2, POST : [(tab[0] \geq min_3) \wedge (tab[1] \geq min_3) \wedge (tab[2] \geq min_3) \wedge (tab[3] \geq min_3)]\}$	$\{min_2 = tab_0[i_1]\}$
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0], min_0 = tab_0[0], i_0 = 1, min_1 = tab_0[i_0], i_1 = i_0 + 1, min_2 = tab_0[i_1], i_2 = i_1 + 1, [\neg(i_2 \leq tab_0.length - 1)]\}$	$\{i_0 = 1\}$, $\{i_1 = i_0 + 1\}$, $\{i_2 = i_1 + 1\}$

Table 2: Paths and MCSs generated by `LocFaults` for the program *Minimum*.

$\{tab[0] = 3, tab[1] = 2, tab[2] = 1, tab[3] = 0\}$, we illustrated in Figure 8 the initial faulty path (see the colorful path in red) and the deviation for which the postcondition is satisfiable (the deviation and the path above the deviated condition are shown in green).

```
1  class Minimum {
2  /* The minimum in an array
       of n integers*/
3  /*@ ensures
4    @ (\forall int k;(k >= 0
         && k < tab.length)
       ;tab[k] >= min);
5    @*/
6    int Minimum (int[] tab)
       {
7      int min=tab[0];
8      int i = 1;
9      while (i<tab.length-1)
         { /*error, the
         condition should
         be (i<tab.length)
         */
10       if (tab[i]<=min){
11         min=tab[i];
12       }
13       i = i+1;
14     }
15     return min;
16   }
17 }
```

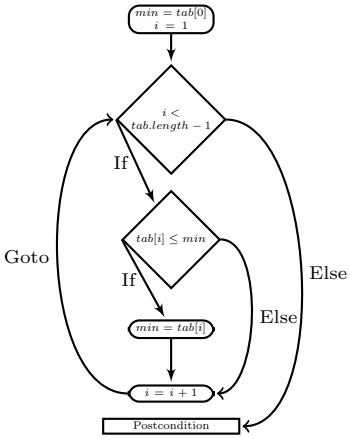


Figure 7: The program Minimum and its normal CFG (non unfolded). The postcondition is $\{\forall$ **int k$; (k \geq 0 \wedge k < tab.length); tab[k] \geq min\}$**

We show in table 2 erroneous paths generated (column $PATH$) and the MCSs calculated (column $MCSs$) for at most 1 condition deviated from the conduct of the counterexample. The first line concerns the path of counterexample; the second for the path obtained by deviating the condition $\{i_2 \leq tab_0.length - 1\}$.

`LocFaults` identifies a single MCS on the path of counterexample that contains the constraint $min_2 = tab_0[i_1]$, the instruction of the line 11 in the second iteration of the loop unfolded. With a deviated condition, the algorithm suspects the third condition of the unfolded loop $i_2 < tab_0.length - 1$; in other words, we need a new iteration to satisfy the postcondition.

This example shows a case of a program with an incorrect loop: the error is on the stopping criterion, it does not allow the program to iterate until the last element of the array input. LocFaults with its deviation mechanism is able to detect this type of error accurately. It provides the user not only suspicious instructions in the loop not unfolded on the original program, but also information about the iterations where they are in the unfolded loop. This information could

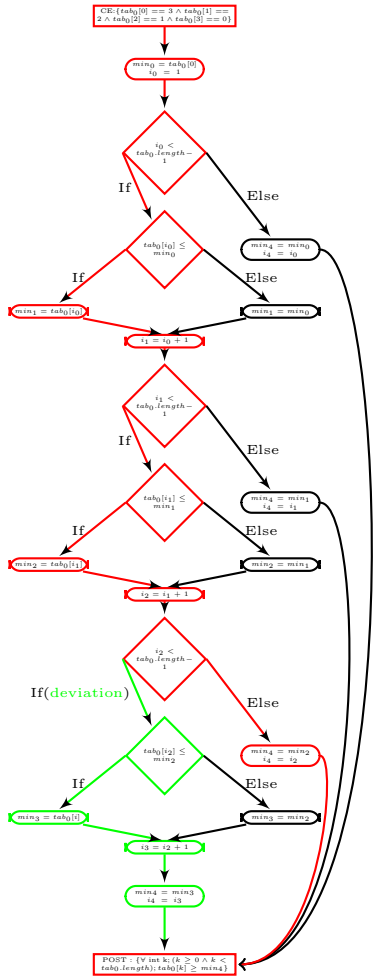


Figure 8: Figure showing the CFG in DSA form of the program *Minimum* by unfolding its loop 3 times, with the path of a counterexample (shown in red) and a deviation satisfying the postcondition (shown in green).

be very useful for the programmer to understand the errors in the loop.

5. ALGORITHM

Our goal is to find MCDs of size less than a bound k ; in other words, we try to give a solution to the problem posed above ($\leq k$-MCD). For this, our algorithm (named `LocFaults`) explores in depth the CFG and generates the paths where at most k conditions are deviated from the conduct of the counterexample.

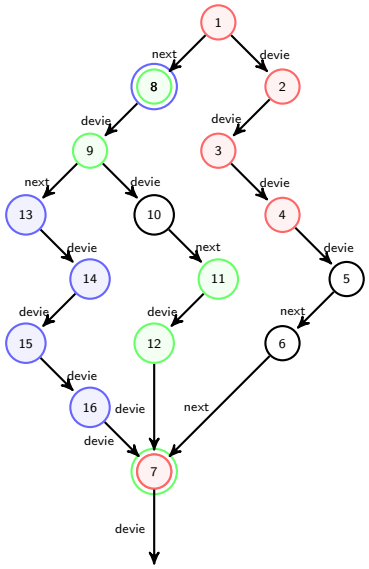
To improve efficiency, our heuristic solution proceeds incrementally. It successively deviates from 0 to k conditions and search the MCSs for the corresponding paths. However, if in step k `LocFaults` deviates a condition c_i and that it has corrected the program, it does not explore in step k' with $k' > k$ paths that involve a deviation from the condition c_i. For this, we add the cardinality of the found minimum deviation (k) as information on the node of c_i.

We will illustrate with an example of our approach, as seen in the graph in Figure 9. Each circle in the graph represents a conditional node visited by the algorithm. The example does not show the block of assignments because we want to

illustrate just how we find the minimal correction deviations of a bounded size as mentioned above. An arc connecting a condition c_1 to another c_2 illustrates that c_2 is reached by the algorithm. There are two ways related to the behavior of the counterexample, where `LocFaults` reaches the condition c_2:

1. by following the branch induced by the condition c_1 ;
2. by following the opposite branch.

The value of the label of arcs for case (1) (resp. (2)) is "*next*" (resp. "*devie*").



the path $< 1, 2, 3, 4, 5, 6, 7, ..., POST >$ is correct
the path $< 1, 8, 9, 10, 11, 12, 7, ..., POST >$ is correct

Figure 9: Figure illustrating the execution of our algorithm on an example in which two minimal correction deviations are detected: $\{1, 2, 3, 4, 7\}$ and $\{8, 9, 11, 12, 7\}$, and one abandoned deviation: $\{8, 13, 14, 15, 16, 7\}$. Knowing that the deviation of the condition "7" has corrected the program for the path $< 1, 2, 3, 4, 5, 6, 7 >$, and for the path $< 1, 8, 9, 10, 11, 12, 7 >$. $POST$ in the figure is the post-condition.

- At the step $k = 5$, our algorithm has identified two MCDs of size equal to 5:
  1. $D_1 = \{1, 2, 3, 4, 7\}$, the node "7" is marked by the value 5 ;
  2. $D_2 = \{8, 9, 11, 12, 7\}$, it was allowed because the value of the marke of the node "7" is equal to the cardinality of D_2.
- At the step $k = 6$, the algorithm has suspended the following deviation $D_3 = \{8, 13, 14, 15, 16, 7\}$, because the cardinality of D_3 is strictly greater than the value of the label of the node "7".

6. PRACTICAL EXPERIENCE

To evaluate the scalability of our method, we compared its performance with that of `BugAssist`[4] on two sets benchmarks[5].

* The first benchmark is illustrative, it contains a set of programs without loops;
* The second benchmark includes 19, 48 and 91 variations for respectively the programs BubbleSort, Sum and SquareRoot. These programs contain loops to study the scalability of our approach compared to `BugAssist`. To increase the complexity of a program, we increase the number of iterations in loops in the execution of each tool; we use the same bound of unfolding loops for `LocFaults` and `BugAssist`.

To generate the CFG and the counterexample, we use the tool CPBPV [8] (Constraint-Programming Framework for Bounded Program Verification). `LocFaults` and `BugAssist` work respectively on Java and C programs. For a fair comparison, we built two equivalent versions for each program:

* a version in Java annotated by a JML specification;
* a version in ANSI-C annotated by the same specification but in ACSL.

Both versions have the same numbers of lines of instructions, including errors. The precondition specifies the counterexample used for the program.

To calculate the MCSs, we used IBM ILOG MIP[6] and CP[7] solvers of CPLEX. We adapted and implemented the algorithm of Liffiton and Sakallah [15], see alg. 1. This implementation takes as input the infeasible set of constraints corresponding to the identified path (C), and b_{mcs}: the bound on the size of calculated MCSs. Each constraint c_i in the system built C is augmented by an indicator y_i for giving $y_i \rightarrow c_i$ in the new system of constraints C'. Assign to y_i the value $True$ implies the constraint c_i; however, assign to y_i value $False$ implies the removal of the constraint c_i. A MCS is obtained by seeking an assignment that satisfies the constraint system with a minimal set of constraints indicators affected with $False$. To limit the number of constraints indicators that can be assigned with $False$, we use the constraint $AtMost(\neg y_1, \neg y_2, ..., \neg y_n, k)$ (see the line 5), the created system is noted in the algorithm C'_k (line 5). Each iteration of the WHILE-loop (lines $6 - 19$) is allowed to find all MCSs of size k, k is incremented by 1 after each iteration. After finding each MCS (lines $8 - 13$), a blocking constraint is added to C'_k and C' to prevent finding this new MCS in the next iterations (lines $15 - 16$). The first loop (lines $4 - 19$) is iterated until all MCSs of C are generated (C' becomes infeasible); it can also stop if the MCSs of size smaller or equal to b_{mcs} are obtained ($k > b_{mcs}$).

[4]The tool BugAssist is available at : `http://bugassist.mpi-sws.org/`
[5]The source code for all programs is available at : `http://www.i3s.unice.fr/~bekkouch/Benchs_Mohammed.html`
[6]IBM ILOG MIP is available at http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
[7]IBM ILOG CP OPTIMIZER is available at http://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/

```
1  Function MCS(C,b_mcs)
   Data: C: Infeasible set of constraints, b_mcs: Integer
   Result: MCS: List of MCSs in C of a cardinality less than
           b_mcs
2  begin
3      C' ← AddYVars(C); MCS ← ∅; k ← 1;
4      while SAT(C') ∧ k ≤ MCS_b do
5          C'_k ← C' ∧ AtMost({¬y_1, ¬y_2, ..., ¬y_n},k)
6          while SAT(C'_k) do
7              newMCS ← ∅
8              forall the indicator y_i do
9                  % y_i indicator of the constraint c_i ∈ C, and
                   val(y_i) is the value of y_i in the solution
                   calculated for C'_k.
10                 si val(y_i) = 0 alors
11                     newMCS ← newMCS ∪ {c_i}.
12                 fin
13             end
14             MCS.add(newMCS).
15             C'_k ← C'_k ∧ BlockingClause(newMCS)
16             C' ← C' ∧ BlockingClause(newMCS)
17         end
18         k ← k + 1
19     end
20     return MCS
21 end
```

Algorithm 1: The algorithm of Liffiton and Sakallah

BugAssist uses the tool CBMC [7] to generate the faulty trace and input data. For Max-SAT solver, we used MSUnCore2 [16].

The experiments were performed with a processor Intel Core i7-3720QM 2.60 GHz with 8 GO of RAM.

6.1 Benchmark without loops

This part serves to illustrate the improvement in LocFaults to reduce the number of subsets of suspects instructions provided to the user: at a given step of the algorithm, the node in the CFG of the program that allows detect a MCD will be marked by the cardinality of the latter; in the next steps, the algorithm will not allow scanning an adjacency list of this node.

Our results[8] show that LocFaults misses errors only for TritypeKO6. While BugAssist misses errors for AbsMinusKO2, AbsMinusKO3, AbsMinusV2KO2 , TritypeKO , TriPerimetreKO, TriMultPerimetreKO and one of two errors in TritypeKO5. The times [9] of our tool are better compared to BugAssist for programs with numerical calculation; they are close for the rest of programs.

We randomly take three programs as examples. And we consider the implementation of two versions of our algorithm with and without marking nodes named respectively LocFaultsV1 and LocFaultsV2.

- Tables 3 and 4 show respectively the suspects sets and times of LocFaultsV1 ;
- Tables 5 and 6 show respectively the suspects sets and times of LocFaultsV2.

In tables 3 and 5, we display the list of calculated MCSs and MCDs. The line number corresponding to the condition is underlined. Tables 4 and 6 give calculation times: P is the pretreatment time which includes the translation of Java program into an abstract syntax tree with JDT tool (Eclipse Java devlopment tools), as well as the construction of CFG; L is the time of the exploration of CFG and calculation of MCSs.

LocFaultsV2 has significantly reduced the deviations generated and the time summing exploration of the CFG and calculation of MCSs by LocFaultsV1, without losing the error; the localizations provided by LocFaultsV2 are more relevant. The eliminated lines of the table 5 are colored blue in the table3. The improved time are shown in bold in the table 4. For example, for the program TritypeKO2, at step 1 of the algorithm, LocFaultsV2 marks the node of condition 26, 35 and 53 (from the counterexample, the program becomes correct by deviating each of these three conditions). This allows, at step 2, to cancel the following deviations: $\{\underline{26}, \underline{29}\}$, $\{\underline{26}, \underline{35}\}$, $\{\underline{29}, \underline{35}\}$, $\{\underline{32}, \underline{35}\}$. Always in step 2, LocFaultsV2 detects two minimal correction deviations more: $\{\underline{29}, \underline{57}\}$, $\{\underline{32}, \underline{44}\}$, the nodes 57 and 44 will be marked (the value of the mark is 2). At step 3, no deviation is selected; for example, $\{\underline{29}, \underline{32}, \underline{44}\}$ is not considered because its cardinal is strictly superior to the mark value of the node 44.

Program	LocFaults				
	P	L			
		$= 0$	≤ 1	≤ 2	≤ 3
TritypeKO2	0, 471	0, 023	0, 241	2, 529	5, 879
TritypeKO4	0, 476	0, 022	0, 114	0, 348	5, 55
TriPerimetreKO3	0, 487	0, 052	0, 237	2, 468	6, 103

Table 4: Computation time, for the results without marking of nodes in the CFG

Program	LocFaults				
	P	L			
		$= 0$	≤ 1	≤ 2	≤ 3
TritypeKO2	0, 496	0, 022	0, 264	**1,208**	**1,119**
TritypeKO4	0, 481	0, 021	0, 106	**0,145**	**1,646**
TriPerimetreKO3	0, 485	0, 04	0, 255	**1,339**	**1,219**

Table 6: Computation time, for the results with marking of nodes in the CFG

6.2 Benchmarks with loops

These benchmarks are used to measure the scalability of LocFaults compared to BugAssist for programs with loops, depending on the increase of unfolding b. We took three programs with loops : BubbleSort, Sum, and SquareRoot. We have caused the *Off-by-one* bug in each of them. The benchmark for each program is created by increasing the number of unfolding b. b is equal to the number of iterations through the loop in the worst case. We also vary the number of deviated conditions for LocFaults from 0 to 3.

We used the MIP solver of CPLEX for BubbleSort. For Sum and SquareRoot, we collaborate the two solvers of CPLEX (CP and MIP) during the localization process. Indeed, during the collection of constraints, we use a variable to keep the information on the type of building CSP. When LocFaults detects an erroneous path[10] and prior to the calculation of MCSs, it takes the good solver depending on the type of CSP corresponding to this path : if it is non-linear, it uses the CP OPTIMIZER solver; otherwise it uses the MIP solver.

[8]The table that shows the calculated MCSs by LocFaults for the programs without loops are available at http://www.i3s.unice.fr/~bekkouch/Benchs_Mohammed.html#rsb

[9]The tables that give the times of LocFaults and BugAssist for the programs without loops are available at http://www.i3s.unice.fr/~bekkouch/Benchs_Mohammed.html#rsba.

[10]An erroneous path is the one on which we identify MCSs.

Program	Counterexample	Errors	LocFaults			
			$= 0$	≤ 1	≤ 2	≤ 3
TritypeKO2	$\{i = 2, j = 2, k = 4\}$	53	{54}	{54}	{54}	{54}
				{21}	{21}	{21}
				{26}	{26}	{26}
				{35},{27},{25}	{35},{27},{25}	{35},{27},{25}
				{53},{25},{27}	{53},{25},{27}	{53},{25},{27}
					{26, 29}	{26, 29}
					{26, 35},{25}	{26, 35},{25}
					{29, 35},{30},{25},{27}	{29, 35},{30},{25},{27}
					{29, 57},{30},{27},{25}	{29, 57},{30},{27},{25}
				{53},{25},{27}	{32, 35},{33},{25},{27}	{32, 35},{33},{25},{27}
					{32, 44},{33},{25},{27}	{32, 44},{33},{25},{27}
					{32, 44},{33},{25},{27}	{26, 29, 35},{30},{25}
					{32, 44},{33},{25},{27}	{26, 32, 35},{33},{25}
					{32, 44},{33},{25},{27}	{26, 32, 57},{25},{33}
					{32, 44},{33},{25},{27}	{29, 32, 35},{33},{25},{27},{30}
					{32, 44},{33},{25},{27}	{29, 32, 44},{33},{25},{27},{30}
TritypeKO4	$\{i = 2, j = 3, k = 3\}$	45	{46}	{46}	{46}	{46}
				{45},{33},{25}	{45},{33},{25}	{45},{33},{25}
					{26, 32}	{26, 32}
					{29, 32}	{29, 32}
					{45, 49},{33},{25}	{45, 49},{33},{25}
					{45, 53},{33},{25}	{45, 53},{33},{25}
					{45, 53},{33},{25}	{26, 45, 49},{33},{25},{27}
					{45, 53},{33},{25}	{26, 45, 53},{33},{25},{27}
					{45, 53},{33},{25}	{26, 45, 57},{33},{25},{27}
					{45, 53},{33},{25}	{29, 32, 49},{30},{25}
					{45, 53},{33},{25}	{29, 45, 49},{33},{25},{30}
					{45, 53},{33},{25}	{29, 45, 53},{33},{25},{30}
					{45, 53},{33},{25}	{29, 45, 57},{33},{25},{30}
					{45, 53},{33},{25}	{32, 35, 49},{25}
					{45, 53},{33},{25}	{32, 35, 53},{25}
					{45, 53},{33},{25}	{32, 35, 57},{25}
TriPerimetreKO3	$\{i = 2, j = 1, k = 2\}$	57	{58}	{58}	{58}	{58}
				{22}	{22}	{22}
				{31}	{31}	{31}
				{37},{32},{27}	{37},{32},{27}	{37},{32},{27}
				{57},{32},{27}	{57},{32},{27}	{57},{32},{27}
					{28, 37},{32},{27},{29}	{28, 37},{32},{27},{29}
					{28, 61},{32},{27},{29}	{28, 61},{32},{27},{29}
					{31, 37},{27}	{31, 37},{27}
					{34, 37},{35},{27},{32}	{34, 37},{35},{27},{32}
					{34, 48},{35},{32},{27}	{34, 48},{35},{32},{27}
					{34, 48},{35},{32},{27}	{28, 31, 37},{29},{27}
					{34, 48},{35},{32},{27}	{28, 31, 52},{29},{27}
					{34, 48},{35},{32},{27}	{28, 34, 37},{35},{27},{29},{32}
					{34, 48},{35},{32},{27}	{28, 34, 48},{35},{27},{29},{32}
					{34, 48},{35},{32},{27}	{31, 34, 37},{27},{35}
					{34, 48},{35},{32},{27}	{31, 34, 61},{27},{35}

Table 3: MCSs and deviations identified by `LocFaults` for programs without loops, without marking of nodes in the CFG

Program	Counterexample	Errors	LocFaults			
			$= 0$	≤ 1	≤ 2	≤ 3
TritypeKO2	$\{i = 2, j = 2, k = 4\}$	53	{54}	{54}	{54}	{54}
				{21}	{21}	{21}
				{26}	{26}	{26}
				{35},{27},{25}	{35},{27},{25}	{35},{27},{25}
				{53},{25},{27}	{53},{25},{27}	{53},{25},{27}
					{29, 57},{30},{27},{25}	{29, 57},{30},{27},{25}
					{32, 44},{33},{25}, {27}	{32, 44},{33},{25}, {27}
TritypeKO4	$\{i = 2, j = 3, k = 3\}$	45	{46}	{46}	{46}	{46}
				{45},{33},{25}	{45},{33},{25}	{45},{33},{25}
					{26, 32}	{26, 32}
					{29, 32}	{29, 32}
					{29, 32}	{32, 35, 49},{25}
					{29, 32}	{32, 35, 53},{25}
					{29, 32}	{32, 35, 57},{25}
TriPerimetreKO3	$\{i = 2, j = 1, k = 2\}$	57	{58}	{58}	{58}	{58}
				{22}	{22}	{22}
				{31}	{31}	{31}
				{37},{32},{27}	{37},{32},{27}	{37},{32},{27}
				{57},{32},{27}	{57},{32},{27}	{57},{32},{27}
					{28, 61},{32},{27},{29}	{28, 61},{32},{27},{29}
					{34, 48},{35},{32},{27}	{34, 48},{35},{32},{27}

Table 5: MCSs and MCDs identified by `LocFaults` for programs without loops, with marking of nodes in the CFG

For each benchmark, we presented an extract of the table containing the computation time[11] (columns P and L show respectively the time of pretreatment and calculating of MCSs), and the graph which corresponds to the time of calculation of MCSs.

6.2.1 *BubbleSort benchmark*

BubbleSort is an implementation of the bubble sort algorithm. This program contains two nested loops; its average complexity is $O(n^2)$, where n is the size of the table sorted : the bubble sort is considered among the worst sort algorithms. The erroneous statement in the program causes the program to sort input array by considering only its $n-1$ first elements. The malfunction of BubbleSort is due to the insufficient number of iterations performed by the loop. This is due to the faulty initialization of the variable i : i = tab.length - 1; the instruction should be i = tab.length.

Programs	b	LocFaults					BugAssist	
		P	L				P	L
			$= 0$	≤ 1	≤ 2	≤ 3		
V0	4	0.751	0.681	0.56	0.52	0.948	0.34	55.27
V1	5	0.813	0.889	0.713	0.776	1.331	0.22	125.40
V2	6	1.068	1.575	1.483	1.805	4.118	0.41	277.14
V3	7	1.153	0.904	0.85	1.597	12.67	0.53	612.79
V4	8	0.842	6.509	6.576	8.799	116.347	1.17	1074.67
V5	9	1.457	18.797	18.891	21.079	492.178	1.24	1665.62
V6	10	0.941	28.745	29.14	35.283	2078.445	1.53	2754.68
V7	11	0.918	59.894	65.289	74.93	4916.434	3.94	7662.90

Table 7: Computation time for benchmark BubbleSort

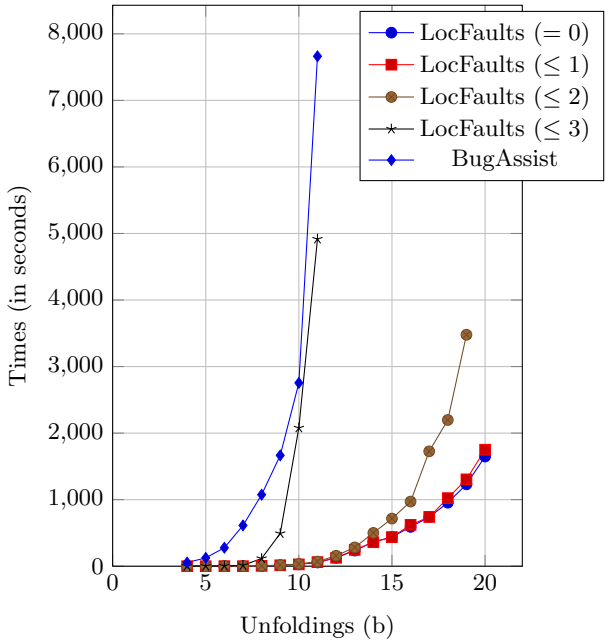


Figure 10: Comparison of the evolution of times of different versions of `LocFaults` and of `BugAssist` for the benchmark BubbleSort, by increasing the unwinding loop limit.

[11]Full tables are available at `http://www.i3s.unice.fr/~bekkouch/Benchs_Mohammed.html#ravb`, the sources of these results are available at `http://www.i3s.unice.fr/~bekkouch/Benchs_Mohammed.html#sr`

MCDs	*MCSs*
$\emptyset$	{5},{6},{9 : 1.11}, {9 : 2.11},{9 : 3.11}, {9 : 4.11},{9 : 5.11},{9 : 6.11},{9 : 7.11},{13}
{9 : 7}	{5},{6},{7},{9 : 1.10},{9 : 2.10},{9 : 3.10}, {9 : 4.10},{9 : 5.10}, {9 : 6.10},{9 : 1.11}, {9 : 2.11},{9 : 3.11},{9 : 4.11},{9 : 5.11}, {9 : 6.11}

Table 8: MCD and MCSs calculated by `LocFaults` for SquareRoot.

The times of `LocFaults` and `BugAssist` for the benchmark BubbleSort are presented in the table 7. The graph illustrates the increase in times of different versions of `LocFaults` and of `BugAssist` depending on the number of unfolding is given in Figure 10.

The runtime of `LocFaults` and of `BugAssist` grows exponentially with the number of unfoldings; the times of `BugAssist` are always the greatest. We can consider that `BugAssist` is ineffective for this benchmark. The different versions of `LocFaults` (with at most 3, 2, 1, and 0 conditions deviated) remain usable up to a certain unfolding. The number of unfolding beyond which growth time of `BugAssist` becomes redhibitory is lower than that of `LocFaults`, that of `LocFaults` with at most 3 conditions diviated is lower than that of `LocFaults` with at most 2 conditions diviated which is also lower than that of `LocFaults` with at most 1 conditions diviated. The times of `LocFaults` with at most 1 and 0 conditions diviated are almost the same.

6.2.2 *SquareRoot and Sum benchmarks*

The program SquareRoot (see fig. 11) permits to find the integer part of the square root of the integer 50. An error is injected at the line 13, which leads to return the value 8; while the program must return 7. This program has been used in the paper describing the approach `BugAssist`, it contains a linear numerical calculation in its loop and nonlinear in its postcondition.

```
1  class SquareRoot{
2    /*@ ensures((res*res<=val) && (res+1)*(res+1)>val);*/
3    int SquareRoot()
4      {
5          int val = 50;
6          int i = 1;
7          int v = 0;
8          int res = 0;
9          while (v < val){
10             v = v + 2*i + 1;
11             i= i + 1;
12         }
13         res = i; /*error: the instruction should be res =
               i - 1*/
14         return res;
15     }
16 }
```

Figure 11: The program SquareRoot

With an unwinding limit of 50, `BugAssist` calculates for this program the following suspicious instructions: $\{9, 10, 11, 13\}$. The time of localization is $36,16s$ and the pretreatment time is $0,12s$.

`LocFaults` displays a suspicious instruction by indicating both its location in the program (instruction line), the line of the condition and the iteration of each loop leading to this instruction. For example, $\{9 : 2.11\}$ corresponds to the instruction that is on line 11 in the program, the latter is in a loop whose line of the stop condition is 9 and the iteration number is 2. The sets suspected by `LocFaults` are provided in the table 8.

The pretreatment time is $0,769s$. The time during the exploration of the CFG and the calculation of MCSs is $1,299s$. We studied the times of `LocFaults` and `BugAssist` of values of *val* ranging from 10 to 100 (the number of unfolding *b* used is equal to *val*), to study the combinatorial behavior of each tool for this program.

Programs	b	LocFaults					BugAssist	
		P	L				P	L
			$= 0$	≤ 1	≤ 2	≤ 3		
V0	10	1.096	1.737	2.098	2.113	2.066	0.05	3.51
V10	20	0.724	0.974	1.131	1.117	1.099	0.05	6.54
V20	30	0.771	1.048	1.16	1.171	1.223	0.08	12.32
V30	40	0.765	1.048	1.248	1.266	1.28	0.09	23.35
V40	50	0.769	1.089	1.271	1.291	1.299	0.12	36.16
V50	60	0.741	1.041	1.251	1.265	1.281	0.14	38.22
V70	80	0.769	1.114	1.407	1.424	1.386	0.19	57.09
V80	90	0.744	1.085	1.454	1.393	1.505	0.22	64.94
V90	100	0.791	1.168	1.605	1.616	1.613	0.24	80.81

Table 9: The computation time for the benchmark SquareRoot

Programs	b	LocFaults					BugAssist	
		P	L				P	L
			$= 0$	≤ 1	≤ 2	≤ 3		
V0	6	0.765	0.427	0.766	0.547	0.608	0.04	2.19
V10	16	0.9	0.785	1.731	1.845	1.615	0.08	17.88
V20	26	1.11	1.449	7.27	7.264	6.34	0.12	53.85
V30	36	1.255	0.389	8.727	4.89	4.103	0.13	108.31
V40	46	1.052	0.129	5.258	5.746	13.558	0.23	206.77
V50	56	1.06	0.163	7.328	6.891	6.781	0.22	341.41
V60	66	1.588	0.235	13.998	13.343	14.698	0.36	593.82
V70	76	0.82	0.141	10.066	9.453	10.531	0.24	455.76
V80	86	0.789	0.141	13.03	12.643	12.843	0.24	548.83
V90	96	0.803	0.157	34.994	28.939	18.141	0.31	785.64

Table 10: The computation time for the benchmark Sum

The program Sum takes a positive integer n from the user, and it calculates the value of $\sum_{i=1}^{n} i$. The postcondition specifies that sum. The error in Sum is in the condition of its loop. It causes to calculate the sum $\sum_{i=1}^{n-1} i$ instead of $\sum_{i=1}^{n} i$. This program contains linear numerical instructions in the core of the loop, and a nonlinear postcondition.

The results in time for SquareRoot and Sum benchmarks are shown in the tables respectively 9 and 10. We also designed the graph that corresponds to the result of each benchmark, see respectively the graphs in Figure 12 and 13. The execution time of `BugAssist` grows rapidly; the times of `LocFaults` are almost constant. The times of `LocFaults` with at most 0, 1, and 2 conditions deviated are similar to those of `LocFaults` with at most 3 conditions deviated.

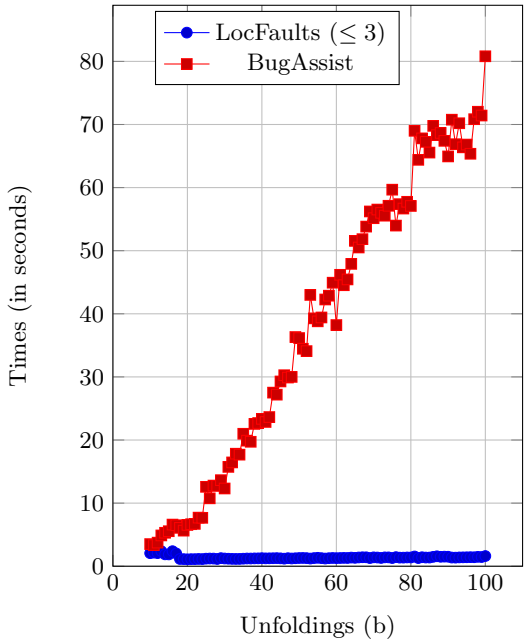


Figure 12: Comparison of the evolution of times of `LocFaults` with at most 3 conditions deviated and of `BugAssist` for the benchmark SquareRoot, by increasing the unwinding loop limit.

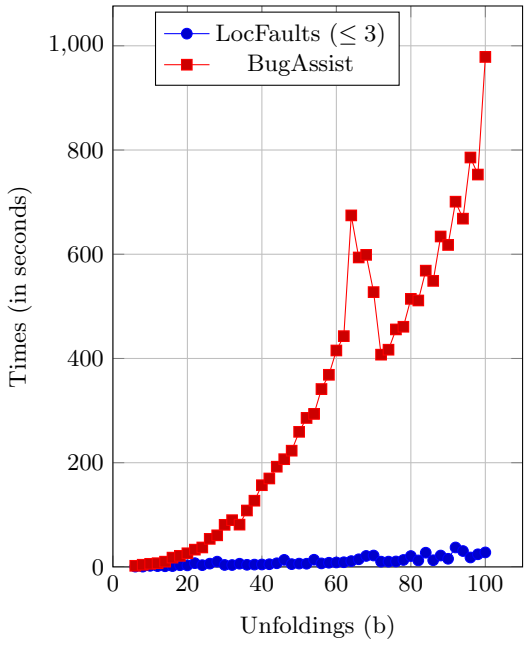


Figure 13: Comparison of the evolution of times of `LocFaults` with at most 3 conditions deviated and of `BugAssist` for the benchmark Sum, by increasing the unwinding loop limit.

7. CONCLUSION

The method `LocFaults` detects the suspicious subsets by analyzing the paths of the CFG to find the MCDs and MCSs from each MCD; it uses constraint solvers. The method `BugAssit` calculates the merger of MCSs of the program by transforming the whole program into a Boolean formula; it uses Max-SAT solvers. Both methods work by starting from a counterexample. In this paper, we presented an exploration of scalability of `LocFaults`, particularly on the treatment of loops with the *Off-by-one* bug. The first results show that `LocFaults` is more effective than `BugAssist` on programs with loops. The times of `BugAssist` rapidly increase with the number of unfolding.

As part of our future work, we plan to validate our results on programs with more complex loops. We envisage to compare the performance of `LocFaults` with existing statistical methods. To improve our tool, we develop an interactive ver-

sion that provides the suspect subsets, one after the other : we want to take advantage of the user's knowledge to select the conditions that should be deviated. We also reflect on how to extend our method to treat numerical instructions with calculation on floating-point.

8. ACKNOWLEDGMENTS

Thanks to Bertrand Neveu for his careful reading and helpful comments on this paper. Thanks to Michel Rueher and Hélène Collavizza for their interesting remarks. Thanks to You Li for his remarks on English mistakes.

9. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007, pages 89–98. IEEE, 2007.

[2] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In ACM SIGSOFT Software Engineering Notes, volume 31, pages 82–87. ACM, 2005.

[3] M. Bekkouche. Bug stories. In `http://www.i3s.unice.fr/~bekkouch/Bug_stories.html`, 2015.

[4] M. Bekkouche, H. Collavizza, and M. Rueher. Une approche csp pour l'aide à la localisation d'erreurs. arXiv preprint arXiv:1404.6567, 2014.

[5] M. Bekkouche, H. Collavizza, and M. Rueher. Locfaults: A new flow-driven and constraint-based error localization approach*. In SAC'15, SVT track, 2015.

[6] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, pages 595–604. IEEE, 2002.

[7] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In Tools and Algorithms for the Construction and Analysis of Systems, pages 168–176. Springer, 2004.

[8] H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv: a constraint-programming framework for bounded program verification. Constraints, 15(2):238–264, 2010.

[9] V. D'silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 27(7):1165–1178, 2008.

[10] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pages 273–282. ACM, 2005.

[11] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In Proceedings of the 24th international conference on Software engineering, pages 467–477. ACM, 2002.

[12] M. Jose and R. Majumdar. Bug-assist: assisting fault localization in ansi-c programs. In Computer Aided Verification, pages 504–509. Springer, 2011.

[13] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. ACM SIGPLAN Notices, 46(6):437–446, 2011.

[14] K.-M. Leung. Debugging loops. In `http://cis.poly.edu/~mleung/CS1114/s08/ch02/debug.htm`.

[15] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning, 40(1):1–33, 2008.

[16] J. Marques-Silva. The msuncore maxsat solver. SAT, page 151, 2009.

[17] Wikipedia. List of software bugs — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=List_of_software_bugs&oldid=648559652`, 2015. [Online; accessed 3-March-2015].

[18] W. E. Wong and V. Debroy. A survey of software fault localization. Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45, 9, 2009.